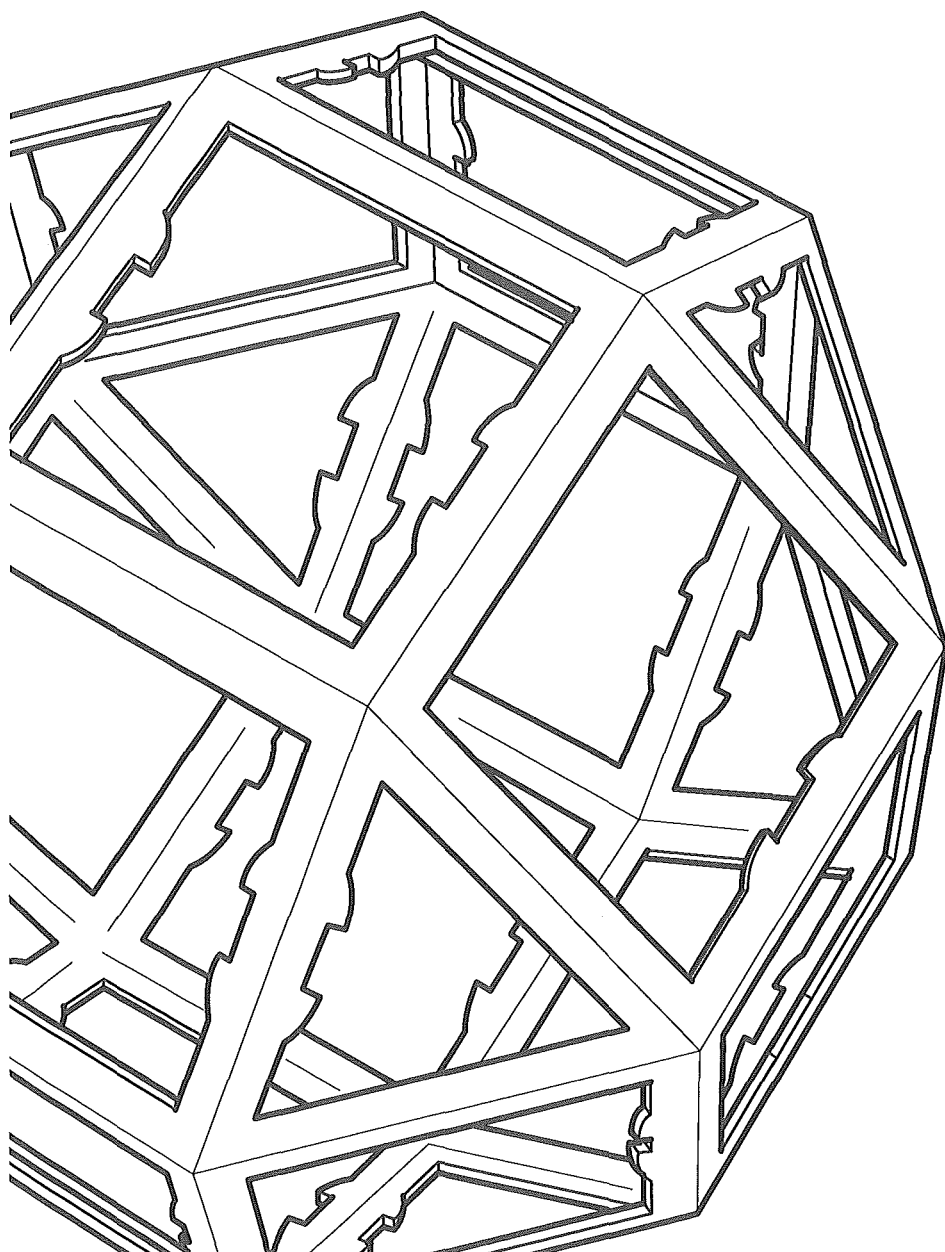


Department of Mathematics and Statistics
College of Engineering

Summer Research Project

Numerical Methods for Shared Memory Parallel Computing

by T.A. Steinke

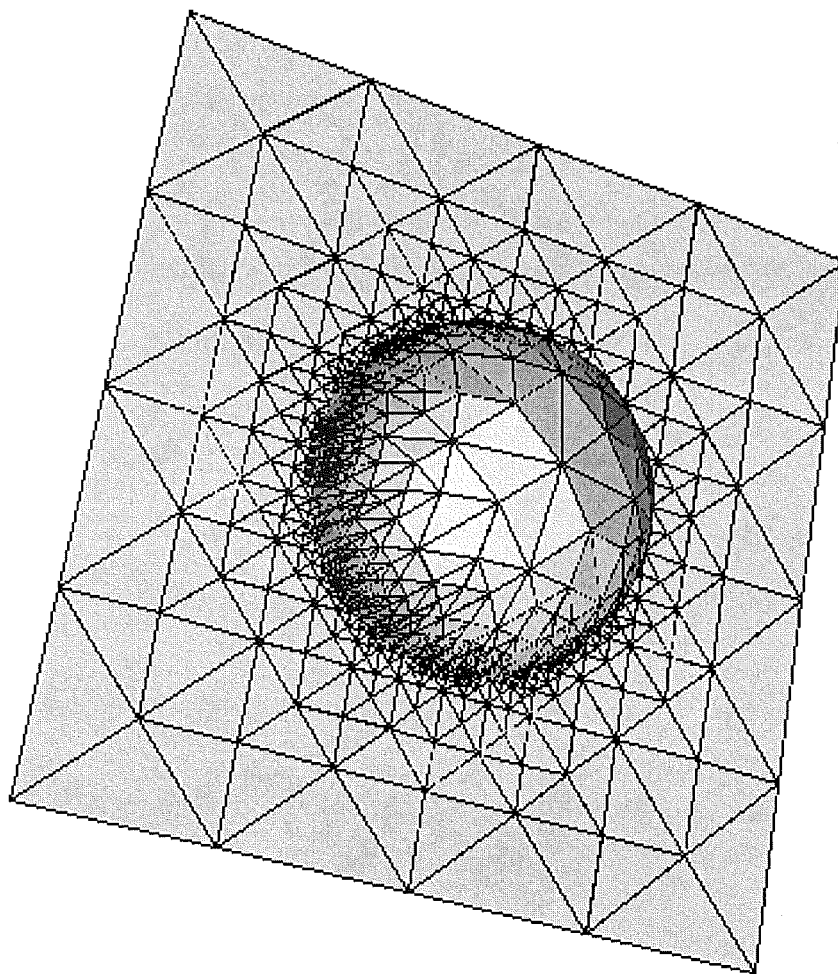


08

MATH305 Summer Research Project:
Numerical Methods for Shared Memory Parallel Computing

T. A. Steinke
Supervisor: R. Beatson

University of Canterbury, New Zealand



1 Abstract

This report discusses shared memory parallel algorithms. It explains the benefits and difficulties of parallelizing algorithms by means of some examples. The main examples are parallel algorithms for calculating a Cholesky decomposition, performing forward and back substitution and adaptively building binary triangle trees.

Contents

1	Abstract	1
2	Introduction	1
3	Parallel Computing Fundamentals	2
3.1	Issues for Parallel Algorithms	3
3.1.1	Synchronization	3
3.1.2	Performance	4
3.1.3	Race Conditions	6
3.2	Methods for Implementing Parallel Algorithms	7
4	Parallel Linear Algebra Algorithms	8
4.1	Cholesky Decomposition	8
4.2	Forward and Back Substitution	10
5	Binary Triangle Trees	15
5.1	Adaptively Constructing a Binary Triangle Tree	16
5.2	A Parallel Algorithm for Adaptive Binary Triangle Tree Construction	18
5.2.1	Performance of the Parallel Algorithm	21
6	Conclusion	22
7	Bibliography	23
8	Appendix	24
8.1	Join Function	24
8.2	Triangle Strips and Fans	25

2 Introduction

Parallel algorithms are becoming an increasingly important issue. Modern trends in computer architecture are leading to an increase in the number of parallel systems in use. Multi-core processors are now available for less than US\$200 ([Intel, 2007]), making them available for everyday use. As such, parallel algorithms need to be developed for traditionally serial tasks to take advantage of this computing revolution. This report deals with such shared memory parallel computers.

There are several tasks which are inherently parallel, such as running several different jobs simultaneously. This report does not deal with such tasks, instead it deals with tasks which are naturally suited to serial execution and require additional work to be made to run in parallel. This report only focuses on computers with up to eight cores, massively parallel systems are beyond the scope of this discussion.

Parallelism has its limitations. Not everything can be ‘perfectly’ parallelized; some algorithms will show very different levels of performance improvement to others. Some algorithms may show significant improvements from parallelization, while others receive no performance increase and, others still, may slow down.

This report attempts to demonstrate what can be parallelized, how it can be done and how well it will perform by means of some examples. It aims to provide a measure of the limitations of parallelism while at the same time explaining the concepts involved and giving some useful examples of parallelization.

Good - more refs/background/previous?

3 Parallel Computing Fundamentals

Parallel computing is an extension of serial computing. Algorithms need^{to} be modified in order to take advantage of what parallelism offers. These parallel algorithms will, if well parallelized, run significantly faster than their serial counterparts. However, significant performance gains are often difficult to achieve. This section of the report discusses the issues involved in parallelization.

refs?

Serial systems perform instructions (the steps comprising the overall method) in a strict order, as defined by the algorithm, one after the other¹. Parallel execution, or, more specifically, multiple-instruction-multiple-data parallel execution, differs from serial execution in that the instructions are not performed in a fixed one-at-a-time order.

A parallel system is comprised of a number of threads working together. A thread is analogous to a serial system; it independently performs instructions and has some of its own resources. However, in the context of a parallel system a thread is only part of a whole; it works in conjunction with other threads to achieve a task. For optimal efficiency, the number of threads should be equal to the number of available cores².

During parallel execution, multiple threads work simultaneously on a single task; therefore multiple instructions will be performed at the same time. Differences between the speeds of the threads will lead to instructions running without a predefined order. The order that instructions are executed in depends on unpredictable factors, such as system load, the results of

¹This ignores instruction-level parallelism such as pipelined and/or superscalar architectures.

²See [Chandra et al., 2001] pages 201 to 204 for an explanation.

resource contentions and random variations in the speeds of the cores the threads are running on. As a result, the order of instructions from different threads is not defined until runtime. Any synchronization between threads must be done explicitly as part of the algorithm.

Shared memory parallel systems, as the name suggests, share access to the same data between the threads. They allow easy communication between threads and are easy for the algorithm designer to understand and implement. An alternative is distributed parallel systems, in such a system, each thread has its own data that it has exclusive access to; communication between threads must be done explicitly through message passing. There are also other systems that are a combination of the above (see [Chandra et al., 2001] pages 204 to 207 for some examples).

Shared memory systems are limited in terms of their scalability³; this is a result of a variety of factors inherent in the shared nature of the system's memory. Distributed systems are more scalable as they do not have any inherent 'bottleneck sharing,' but they are more difficult to program. This report deals with shared memory systems only.

Parallelization involves modifying an algorithm so that independent sub-tasks, which can be run simultaneously in different threads without interference, are distributed amongst several threads. This means that these independent sub-tasks need to be identified and isolated. In the process of doing so, other issues may be raised which will be discussed later. Parallelization is often difficult to achieve easily and efficiently, as such, parallelizing algorithms is a discipline unto itself.

3.1 Issues for Parallel Algorithms

There are a number of important issues for parallel algorithms. Below is a brief discussion of some of them.

3.1.1 Synchronization

Almost all parallel programs will require some form of synchronization between threads. The following examples demonstrate the need for the two most common forms:

- Consider a two part algorithm, where each part is comprised of independent sub-tasks. The sub-tasks can be run in parallel, but the algorithm requires that the first part be completed before the second begins. The program will need to ensure all threads complete their portions of the first part before continuing to the next part.

³See [Pressel et al., 1999] for a study of the limits, in terms of scalability, of shared memory architectures.

- Consider an algorithm in which there is a set of independent tasks that need to be completed; threads remove tasks from the set, complete them and possibly add new tasks to the set. The program must ensure that no more than one thread is trying to add or remove a task to or from the set at any time.

The first example requires a **barrier** - a synchronization construct which makes threads wait at the barrier until all threads arrive, at which point they all continue. A barrier ensures that all threads have performed the preceding tasks before continuing.

The second example requires a **mutex**, also known as a lock. A mutex (originating from the term mutual exclusion) is a synchronization construct that can be acquired and released. Only one thread can hold a mutex at any given time. If a thread attempts to acquire a mutex currently held by another thread then it is forced to wait until that thread releases it. As a result, a mutex can be used to protect data. One must ensure that threads only modify critical data while they are holding the mutex for that data. Since only one thread can hold the mutex at any one time, only one thread will access the data at any one time.

3.1.2 Performance

The primary reason for writing parallel algorithms is to improve performance. Unfortunately, parallelizing an algorithm is not guaranteed to make the algorithm run faster, in fact, it may even slow it down⁴. The primary reason for this is parallel overhead - the additional cost incurred from needing to manage multiple threads.

Parallel algorithm designers must be aware of not only the factors which affect the performance of serial algorithms, but also of factors which specifically affect parallel algorithms.

An unavoidable overhead incurred by parallel algorithms is the cost of starting multiple threads at the beginning of the parallel part of an algorithm and then stopping them at the end. However, one should minimize the number of times this must happen. Consider the following pair of nested loops:

```

1      for i = 1 to n
2          for j = 1 to n
3              do_work()
4          end
5      end
```

⁴An example of this being the failed attempt at direct parallelization of the adaptive binary-triangle-tree-building algorithm to be discussed later.

If the iterations of the outer loop (line 1) were split amongst the threads, then the threads would have to be started once and stopped once. However, if the iterations of the inner loop (line 2) were split amongst the threads, then the threads would have to be started and stopped for every iteration of the outer loop. Simply by the choice of which loop to parallelize, the parallel overhead can change by a factor of n . This demonstrates the effect of granularity - the size of the sub-tasks that are to be done in parallel. Larger sub-tasks (coarser granularity) incur a lower parallel overhead than smaller sub-tasks (finer granularity).

where $n =$

Another overhead is synchronization. Acquiring and releasing mutexes and waiting at barriers takes time and is best avoided. According to [Chandra et al., 2001]⁵, executing⁶ a barrier with eight cores takes approximately 1,000 processor cycles, and acquiring and releasing a mutex once per thread takes approximately 11,000 processor cycles. It was also noted that the time for the mutex grows quadratically with the number of threads.

A very important limiting factor affecting performance is the proportion of the algorithm which runs in parallel. Most, if not all, parallel algorithms will contain parts that need to be run serially. If this serial part of the algorithm accounts for a significant portion of the work, then this limits the advantages of parallelism.

In general, if f is the fraction of the work which can be done in parallel (and $(1 - f)$ is the fraction that must be done serially) and S_p is the speedup⁷ achieved with p threads, then the time taken, $t(p)$, is given by Amdahl's law:

$$t(p) = ((1 - f) + \frac{f}{S_p})t(1),$$

where $t(1)$ is the time taken to run serially. One can clearly see from the above equation that, as more and more cores are used, the serial portion of the algorithm takes up a larger and larger portion of the running time. No matter how many cores are available, the total time will never fall below the time required for the serial portion.

Usually, the 'best case' speedup is linear ($S_p = p$). In practice, linear speedup usually⁸ forms an upper bound. This is because of factors such as the cost of initializing each thread, other per-thread operations and the time required for various means of inter-thread communication and synchronization.

⁵See pages 172, 192 and 196.

⁶The tests were conducted on an SGI Origin 2000 with sixteen 19MHz MIPS R10000 microprocessors, each containing a 32KB on-chip data cache and a 4MB external cache. Compiled using the MIPSPro 7.2.1 Fortran compiler with OpenMP.

⁷Speedup is defined as the number of times faster the algorithm runs with the given number of processors than when run serially.

⁸Super-linear speedups can be achieved from effects such as increased caching from multiple cores. See [Chandra et al., 2001] pages 186 to 188 for an example.

By the definition of an algorithm⁹, there must be a finite number of instructions to execute; a processor core can only execute an integral number of instructions. Therefore, the instructions can only be distributed amongst a finite number of cores. So, there is an upper limit to the number of processor cores that can be used effectively; this also puts a theoretical limit on the potential parallel gain.

Load balancing is another important factor in parallel algorithm performance. If the work is divided unevenly between threads, then some threads will take much longer to perform their tasks; this usually leads to the other threads being forced to wait idly for the slow threads to complete. How work is divided amongst threads is different from algorithm to algorithm; however, it is important to take load balancing into consideration when designing a parallel algorithm.

3.1.3 Race Conditions

An important issue is ensuring that parallel sections of the algorithm do not interfere with one another. Problems occur when some data is being modified by a thread while, at the same time, another thread is trying to access it. When such a situation occurs, it is called a race condition.

Race conditions cause problems because data may enter a transitive state while it is currently being modified by a thread. If another thread tries accessing data that is currently in a transitive state, then incorrect and unpredictable behavior may occur. It is, therefore, important to ensure that there is no possibility of a thread accessing data that is currently being modified. There are three means of achieving this:

- The simplest option is to give threads exclusive access to data through a mutex. However, a mutex will slow down the algorithm. This is because there is a certain amount of work associated with obtaining and releasing a mutex; also, a previously parallel task must now be executed serially, meaning that parallel performance gains are lost for this section of the algorithm. This is an effective and simple solution, but generally the worst solution in terms of performance.
- Another approach is to make a copy of the data in question for each thread and ensure that each thread only accesses its own copy. This approach is often easy to implement and usually the best solution if available. Unfortunately, this is not always feasible; if the data in question is needed for inter-thread communication, then a per-thread private copy will not work.

⁹The 'generally accepted' definition, and the one used here, being a method for solving a given problem in a finite number of well defined steps.

When this fails, creative adaptations are possible, for example, one could make one copy of the data for each pair of threads (along with a mutex) so that only two threads will try accessing the shared data rather than all of them. This still allows inter-thread communication but reduces mutex contention.

- The last method of avoiding race conditions is to restructure the algorithm to avoid the problem altogether. This is not always possible and depends on the algorithm in question.

3.2 Methods for Implementing Parallel Algorithms

There are a number of ways of implementing parallel algorithms, the best choice is dependent on the nature of the algorithm as well as other external factors. This section provides a brief overview of the variety of methods available.

Most parallel algorithms fall into one of two categories:

- A pipeline involves threads linked into a chain, much like an assembly line. Tasks enter the pipeline at one end and each thread processes part of the task and passes it on to the next thread until the task reaches the end and is finished. This can be an effective parallel algorithm. However, usually the steps in the chain are well defined and, therefore, the number of steps and, hence, the number of threads is fixed.
- A thread pool involves a task being partitioned amongst multiple threads, which then simultaneously work on their part of the task. This approach leads to a flexible number of threads, as normally the number of partitions can be altered accordingly. This allows the number of threads to be adjusted according to the number of cores available in order to achieve higher performance.

Implementing a parallel algorithm requires a means of creating and managing several threads. There are various ways in which this can be achieved, the most common methods are through library functions and through compiler directive extensions:

- Library functions - a set of commands to create, synchronise, coordinate and destroy threads - provide full control over the threads to the user. This is useful when a specific number of threads is required, such as in a pipeline. However, managing threads in this manner is additional work that is best avoided. An example of a threading library is the POSIX Pthread library.
- Compiler directive extensions provide a powerful and easy means of expressing parallelism. Compiler directives form an extension to the programming language, they add instructions

that start, manage and stop threads. Compiler directives differ from library functions in that much of the thread management is done automatically. The implementation is not required to decide upon the number of threads, explicitly start them, stop them or divide the work amongst them. Simple directives can start several threads, partition the iterations of a loop amongst the threads and then stop the threads. This is very effective for a thread pool type algorithm, where often only a small number of directives are needed to effectively implement the algorithm. An example of a compiler directive extension is OpenMP.

4 Parallel Linear Algebra Algorithms

Linear algebra is an important part of scientific computing. It has many applications in the fields of mathematics, science and engineering. Linear systems can become quite large and time consuming to solve, as such, there is a strong demand for parallel linear algebra algorithms. Parallel algorithms for solving linear systems can be simple and efficient, as the examples demonstrate.

4.1 Cholesky Decomposition

The Cholesky decomposition is a method for creating the LU-decomposition for a symmetric positive definite matrix¹⁰. This problem arises as a critical sub-task in many applications such as data fitting, nonlinear optimization and finding numerical solutions to partial differential equations. This means finding L for a given symmetric positive definite $n \times n$ matrix A such that:

$$A = LL^T,$$

where L is a lower triangular $n \times n$ matrix.

The algorithm below is a parallel Cholesky decomposition algorithm. It assumes that only the lower half of A is stored in memory (since A is symmetric) and over-writes the storage with L . This algorithm is based on what appears in [Shonkwiler and Lefton, 2006] on page 169 and in [Golub and van Loan, 1996] on page 304.

```

1      for k = 1 to n
2           $A_{(k,k)} \leftarrow \sqrt{A_{(k,k)}}$ 
3          for i = k + 1 to n (in parallel)
```

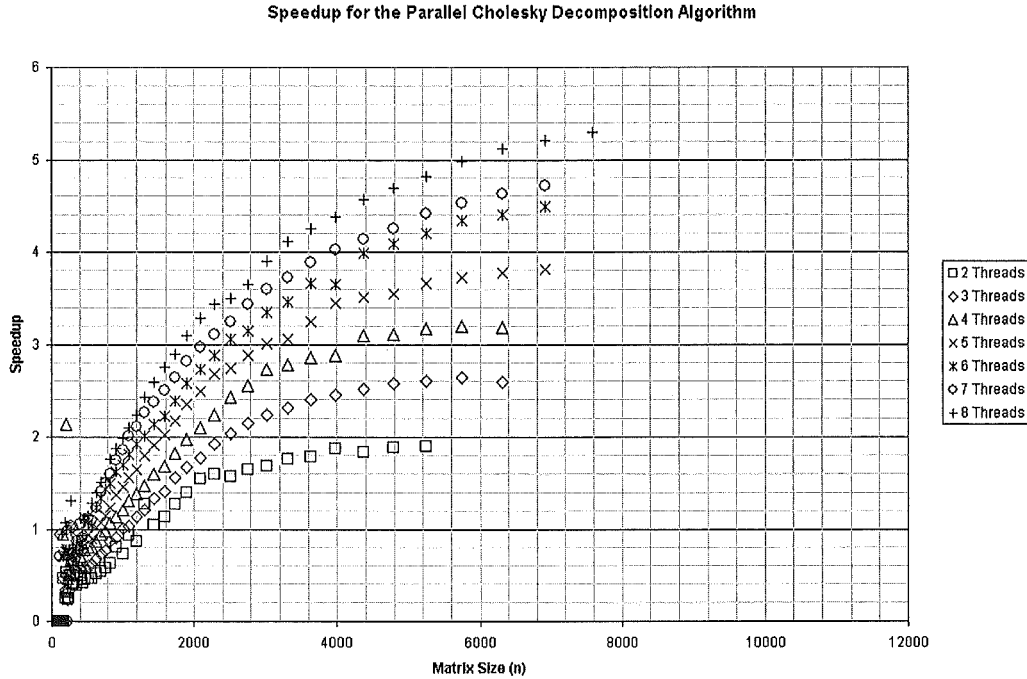
¹⁰This report only deals with real matrices. However, the Cholesky decomposition is also applicable to Hermitian matrices.

```

4          $A_{(i,k)} \leftarrow A_{(i,k)} / A_{(k,k)}$ 
5     end (barrier)
6     for i = k + 1 to n (in parallel)
7         for j = k + 1 to i
8              $A_{(i,j)} \leftarrow A_{(i,j)} - A_{(i,k)} A_{(j,k)}$ 
9         end
10    end (barrier)
11 end

```

This algorithm was implemented using C and OpenMP, compiled with the Intel C++ Compiler and run on an eight core Intel Quad-Core Xeon E5335 (2GHz) computer. The following graph illustrates the speedup achieved by the above algorithm:



It is clear that the parallel algorithm achieves a significant speedup. The algorithm runs more than five times as fast using eight processors than it does using one processor for matrices exceeding a size of 6000×6000 . However, it should be noted that this is a sub-linear speedup, as it is not quite eight times as fast. This can be explained by various reasons, including the synchronization overhead incurred at the end of the parallelized loops, the limited memory bandwidth and the cost of creating multiple threads n times.

The reason this algorithm parallelizes so well is that this is an $O(n^3)$ algorithm and the threads need to be synchronized $O(n)$ times. This means that the parallel overhead grows linearly with respect to matrix size, whereas, the total running time increases cubically. The ratio of actual computing time to parallel overhead increases quadratically with the matrix size; therefore, as the matrix size becomes sufficiently large, the parallel overhead takes up an insignificantly small

portion of the running time of the algorithm. This results in the high speedup observed.

Even greater speedup might be observed if the outermost loop (line 1) were run in parallel rather than the second level loops (lines 3 and 6). Unfortunately, running the outermost loop in parallel would result in an incorrect algorithm. This is because of a data dependency between iterations of the outermost loop; each iteration of the outermost loop assumes that the previous iterations have already been completed. If the outermost loop were run in parallel there would be no means of ensuring that the previous iterations have already been completed and there would be no way of knowing if the algorithm would produce correct results.

Parallelizing and implementing this algorithm is relatively easy, it merely involves distributing the loop iterations amongst the threads. Using OpenMP this can be achieved using only a few lines of code. However, one must be careful of load balancing issues in the second parallel loop (line 6); the work per iteration increases with i as the inner loop (line 7) must perform more iterations for larger values of i . If the iterations are evenly divided into contiguous blocks, then the thread with the last block has significantly more to do than the other threads. In this situation the slow thread will finish last and the other threads will be forced to wait idly at the implicit barrier at the end of the loop, this is inefficient, as a better distribution strategy would result in less idle waiting and a faster algorithm.

4.2 Forward and Back Substitution

Forward and back substitution are methods used to solve lower and upper triangular linear systems. It arises as a critical sub-problem in almost all common methods for solving linear systems.

One situation where triangular systems need to be solved is when solving symmetric positive definite systems. In this case, first the Cholesky decomposition of the coefficient matrix can be calculated (see Section 4.1), then the problem can be converted into two triangular systems as follows:

$$\begin{aligned}
 Ax &= \mathbf{b} \\
 A &= LL^T \\
 \Rightarrow LL^T \mathbf{x} &= \mathbf{b} \\
 L^T \mathbf{x} &= \mathbf{y} \\
 \Rightarrow Ly &= \mathbf{b}.
 \end{aligned}$$

One must first solve the lower triangular system $Ly = \mathbf{b}$, then the upper triangular system $L^T \mathbf{x} = \mathbf{y}$ to obtain \mathbf{x} .

The following are four parallel forward and back substitution algorithms. The input is L or U , a lower or upper triangular $n \times n$ matrix, and \mathbf{b} , an $n \times 1$ column vector. The output is \mathbf{x} , another $n \times 1$ column vector, such that $L\mathbf{x} = \mathbf{b}$ or $U\mathbf{x} = \mathbf{b}$. Note that the storage space for \mathbf{b} is over-written with \mathbf{x} , and in the row version, a sum reduction¹¹ must be carried out on the *sum* variable when the algorithm is run in parallel.

```

1      (Forward Substitution, Row Version)
2      for i = 1 to n
3          sum ← 0
4          for j = 1 to i - 1 (in parallel)
5              sum ← sum +  $L_{(i,j)}$   $\mathbf{b}_j$ 
6          end (barrier)
7           $\mathbf{b}_i \leftarrow (\mathbf{b}_i - \text{sum}) / L_{(i,i)}$ 
8      end

```

```

1      (Forward Substitution, Column Version)
2      for i = 1 to n
3           $\mathbf{b}_i \leftarrow \mathbf{b}_i / L_{(i,i)}$ 
4          for j = i + 1 to n (in parallel)
5               $\mathbf{b}_j \leftarrow \mathbf{b}_j - L_{(j,i)} \mathbf{b}_i$ 
6          end (barrier)
7      end

```

```

1      (Back Substitution, Row Version)
2      for i = n to 1
3          sum ← 0
4          for j = n to i + 1 (in parallel)
5              sum ← sum +  $U_{(i,j)}$   $\mathbf{b}_j$ 
6          end (barrier)
7           $\mathbf{b}_i \leftarrow (\mathbf{b}_i - \text{sum}) / U_{(i,i)}$ 
8      end

```

¹¹A sum reduction means that each thread independently computes the sum for its portion of the loop, and at the end of the loop all the sums from the different threads are added together to get the overall sum. Similar reductions can be preformed for other operations such as multiplication, maximum and minimum. More information on reductions can be found in [Chandra et al., 2001].

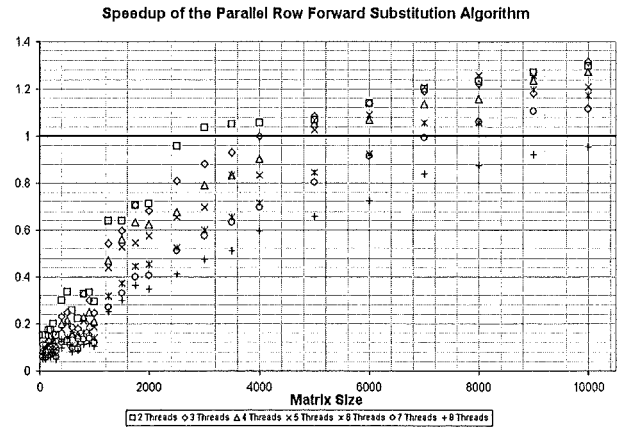
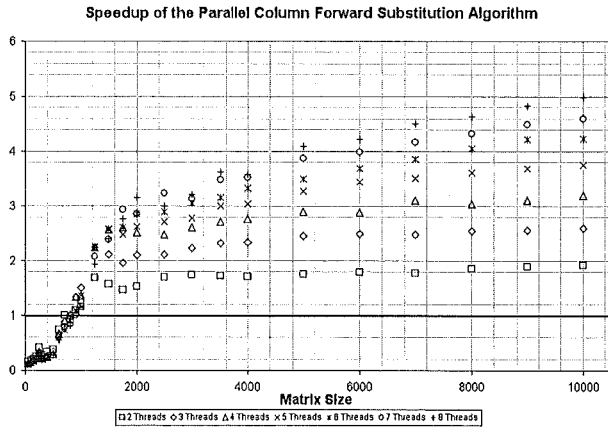
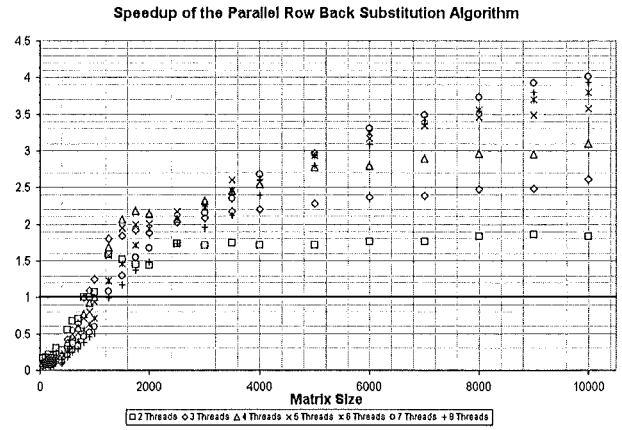
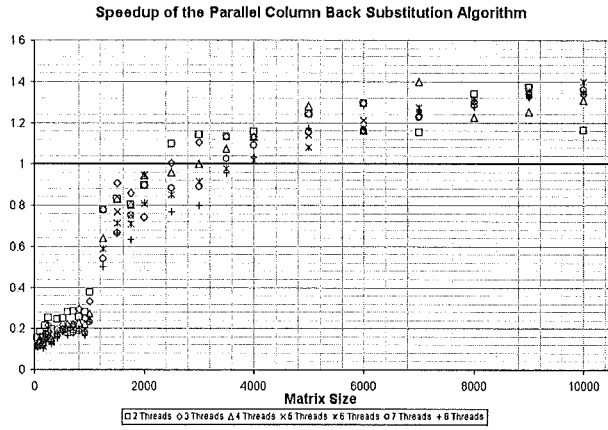
```

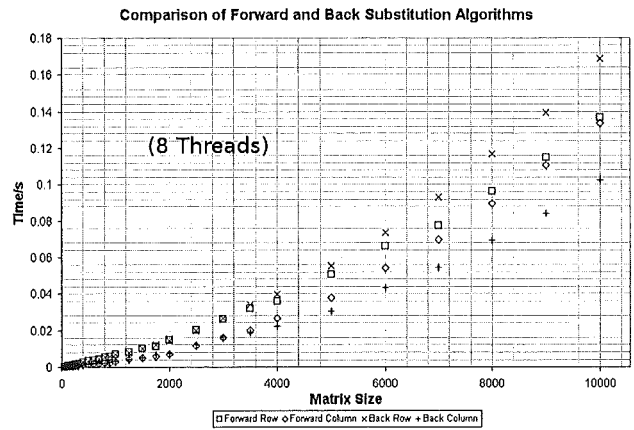
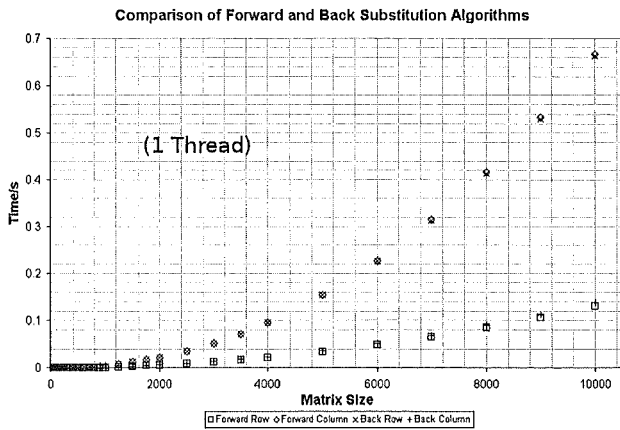
1      (Back Substitution, Column Version)
2      for i = n to 1
3           $b_i \leftarrow b_i / U_{(i,i)}$ 
4          for j = i - 1 to 1 (in parallel)
5               $b_j \leftarrow b_j - U_{(j,i)} b_i$ 
6          end (barrier)
7      end

```

These algorithms are based on [Shonkwiler and Lefton, 2006] pages 127 to 134.

The algorithms were implemented in C with OpenMP, compiled with the Intel C++ Compiler and run on an eight core Intel Quad-Core Xeon E5335 (2GHz) computer. The algorithms were used to solve a pair of systems created by a Cholesky decomposition. The results of these tests are illustrated in the following graphs.





The above graphs clearly show three important characteristics, namely:

- The row back substitution and column forward substitution algorithms parallelize very well, almost as well as the Cholesky decomposition from earlier, whereas the other two do not.
- The column back substitution and the row forward substitution perform significantly better than their counterparts when run with a single thread. However, this difference significantly diminishes with eight threads.
- There is a sharp change in gradient of the speedup just before the matrix size reaches 2000.

The reason for this is caching, which is a feature of modern computer hardware. It involves using a small amount of high speed memory, called a cache, to store important parts of the main memory, which is slower. The effect of this is that accessing data stored in the cache is an order of magnitude faster than accessing data that is not in the cache. Normally, one or more caches will be associated with each core and there will be shared caches as well.

The usefulness of a cache is dependant on what data is stored in it. If the data in the cache is rarely used the cache will not improve performance significantly. However, if the data in the cache is regularly used, then performance may increase dramatically. Most caches are designed to store data that has been recently used and other ‘close’ by data. ‘Close,’ in this context, means being stored in memory addresses, which are numbers, that are similar in value. See [Tanenbaum and Goodman, 1998] for further information on caching.

Caches will improve performance when either the same data is accessed repeatedly or data is accessed ‘close’ to recently accessed data. This means that, if data is read sequentially (accessed in the order it appears in memory), access will be faster than if data were accessed in a non-sequential manner.

For the implementation of the above algorithms, the matrix L was stored as an array of rows. This means that the rows are contiguous in memory, as such, iterating along a row results in sequential memory access, whereas, iterating along a column does not. For the row forward substitution algorithm the innermost loop (line 4) iterates along rows but in the column version it iterates along columns. This explains why the row forward substitution performed better than the column version when run serially.

For the back substitution algorithms, since they were using L^T which was stored as L , the situation is reversed. Iterating along a row in the transpose becomes iterating along a column of L and vice versa. This explains why the column back substitution performed better than the row version when run in serial.

Since each core has a cache of its own, more caches become available as more cores are used. The increase in the number of available caches will increase performance; this is another advantage of parallelism. This explains the second point, as more cache becomes available the column forward and row back substitution algorithms speed up as a result of faster memory access.

However, the row forward and column back substitution algorithms, which already have fast memory access due to their sequential access to L , experience little further improvement in memory access time due to more caches. As a result, the column forward and row back substitution algorithms parallelize much better than their counterparts, but this parallel speedup can do no more than compensate for the fact that they have poor cache performance.

The last characteristic, the ‘bend’ in the speedup graphs, can be explained by the fact that, for sufficiently small matrices, L is small enough to fit into the cache. This means that all memory access is extremely fast. However, once L becomes too big to fit in the cache, the speedup decreases as more load is placed on the shared main memory.

This demonstrates that caching is an important issue and can have strange effects of parallel algorithms. It also shows that just because an algorithm parallelizes well that does not make it the best algorithm for the task.

One very important thing to note is that the back and forward substitution algorithms do not benefit from parallelism as much as the Cholesky decomposition algorithm does. This is a result of the fact that the algorithms are $O(n^2)$ and require $O(n)$ synchronizations. This means that the parallel overhead involved in synchronization is much larger relative to the work than in the Cholesky decomposition algorithm. This leads to a lower performance improvement from parallelism.

5 Binary Triangle Trees

Binary Triangle Trees are an important data structure. Their primary use is in the field of computer graphics, where they are used to build adaptive triangle meshes. Therefore, constructing them is an important task, a task which could benefit from parallelism. However, the algorithm (explained below) does not parallelize easily. It is included as an example of an algorithm that is difficult to parallelize.

Note that, in the context of this report, a triangle is a two dimensional surface in a three dimensional real space which is formed by the part of a plane bounded by three lines joining three points in that plane. In other words, triangles are to be considered as ‘solid’ rather than ‘wireframe.’ This means that a triangle can be defined by three points, **a**, **b** and **c**, as:

$$\text{conv}\{\mathbf{a}, \mathbf{b}, \mathbf{c}\} = \{(\mathbf{a}t + \mathbf{b}(1-t))s + \mathbf{c}(1-s), s, t \in [0, 1]\}.$$

Triangles are widely used in real-time computer graphics, the reason being their linearity and simplicity. A triangle can be represented by a set of three points. Any affine transformation can be applied to a triangle simply by applying the transformation to the set of points which define that triangle and then redefining the triangle with the transformed points. This means that the transformations involved in computer graphics can be easily applied to triangles. Triangles can also be easily drawn to the graphics output. All of these facts make triangles a very useful object in computer graphics.

General objects are not as easy as triangles to render. This is because affine transformations cannot be easily performed on a general object, neither can the transformed object be easily drawn to the graphics output. As a result, complex objects are often approximated by a large number of triangles, called a triangle mesh, for the purposes of rendering.

A triangle mesh needs to fulfil the following three criteria in order to be useful in computer graphics.

Triangle Mesh Criteria:

1. The triangle mesh needs to be continuous (unless the surface it represents is discontinuous at that point). If it is not continuous, it gives the appearance of ‘cracks’ in the surface where adjacent triangles do not join up.
2. The triangle mesh needs to accurately represent the surface. If it does not it is a poor approximation. (This requires a suitable error metric to determine how accurate the triangle mesh is.)
3. The triangle mesh needs to be comprised of a minimal amount of triangles. If the triangle

mesh contains too many triangles, it requires too much of the computer's resources to display it. As such, large numbers of triangles should not be used where fewer triangles would be sufficient.

Building a triangle mesh which fulfils the above criteria for a given surface is one of the problems that binary triangle trees can help to solve. This report studies binary triangle trees in the context of this problem.

5.1 Adaptively Constructing a Binary Triangle Tree

For the purposes of the algorithm detailed below, the surface is described parametrically by a function $f : [0, 1]^2 \rightarrow \mathbb{R}^3$. Also, there exists an error metric $g : ([0, 1]^2)^3 \rightarrow \mathbb{R}^+$ that determines how accurately a given triangle (represented by its three vertices) approximates the surface. Additionally, an error tolerance, ϵ , needs to be specified such that if $g \leq \epsilon$, then a triangle can be assumed to be sufficiently accurate. If $g > \epsilon$, then several smaller triangles need to be used instead.

The algorithm, based on [Duchaineau et al., 1997], begins by covering the parameter space $([0, 1]^2)$ with two right angled isosceles triangles (see Figure 1), this provides the coarsest approximation. Each triangle is then tested to see if it is sufficiently accurate (by testing if $g \leq \epsilon$); if it is not, then it is divided into two smaller triangles (see Figure 1) and the process is repeated.

It is assumed that eventually the splitting must stop. If this is not the case, then it needs to be done artificially.

When a triangle is divided into two smaller triangles, the hypotenuse is split in half. (See Figure 1.) This creates two new right angled isosceles triangles. This hierarchy of triangles forms the binary triangle tree. The trees start with two triangles (the roots) and then the triangles are repeatedly divided. In a division, the new triangles form the child nodes. This means each node in the tree has either two or zero child nodes and each triangle in the mesh has a place in one of the two binary triangle trees. Note that only the triangles with no children, the 'leaf nodes', are present in the triangle mesh. See Figure 2 for an example of how triangle meshes created in this fashion and binary triangle trees correlate.

Every triangle (except the roots) will be half the size of its parent. The size of triangles can be quantified through levels, with smaller triangles at higher levels. If a triangle is at level n , then its children will be at level $n + 1$ and the root triangles are at level zero. This means that the level of a triangle directly correlates to its height in the binary triangle tree.

The basic algorithm of repeated splitting fulfils criteria 2 and 3 discussed in Section 5. However, this does not ensure continuity in the mesh. Cracks may occur when a so-called 'T-vertex' is

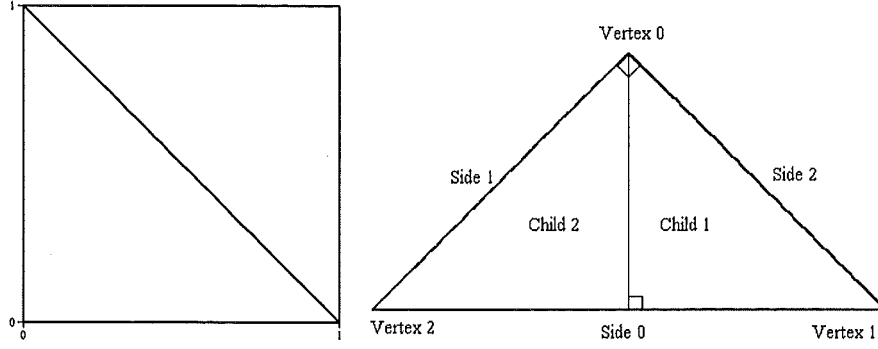


Figure 1: The two base triangles covering the parameter space. (left) Reference diagram of a triangle, with children. (right)

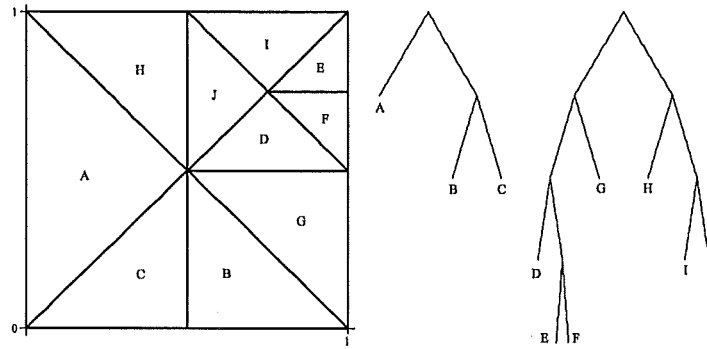


Figure 2: A triangle mesh. (left) The corresponding binary triangle trees. (right)

present. This situation arises whenever a vertex from one triangle lies in the interior of an edge of another triangle. Figure 3 gives an example of a T-vertex and shows how the difference between the linear approximation of the surface along the edge of a triangle and the actual function value at the vertex leads to a crack in the triangle mesh.

Figure 3 shows how the binary-triangle-tree-building algorithm removes T-vertices and, therefore, cracks through additional splitting.

When the algorithm splits a triangle it may need to split the triangle on side 0 (the hypotenuse) first. Splits fall into three categories:

- A 'Single Split' occurs when there is no triangle on side 0 of the original triangle. In this case only one triangle needs to be split.
- A 'Diamond Split' occurs when the triangle on side 0 of the original triangle is at the same level and forms a square with the original triangle. This happens when the triangle on side 0 of the triangle of side 0 of the original triangle is the original triangle. In this

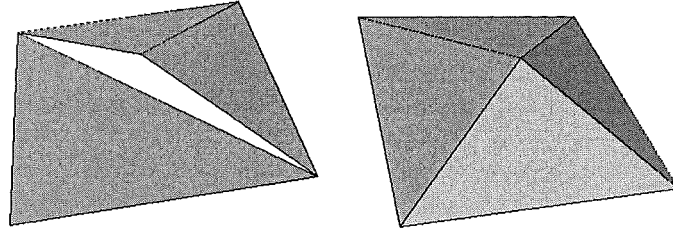


Figure 3: A simple triangle mesh with a crack. (left) The same mesh with the crack removed. (right)

case both the original triangle and the triangle on side 0 need to be split.

- A ‘Split Recursion’ occurs when the triangle on side 0 of the original triangle is at a lower level and is, therefore, larger than it is. This happens when the triangle on side 0 of the original triangle is not the original triangle. In this case the triangle on side 0 of the original triangle needs to be split first (this split once again falls into one of the three categories), then a diamond split needs to happen with the original triangle and one of the child triangles of the triangle that was just split.

Figure 4 shows an example of each of the above three cases.

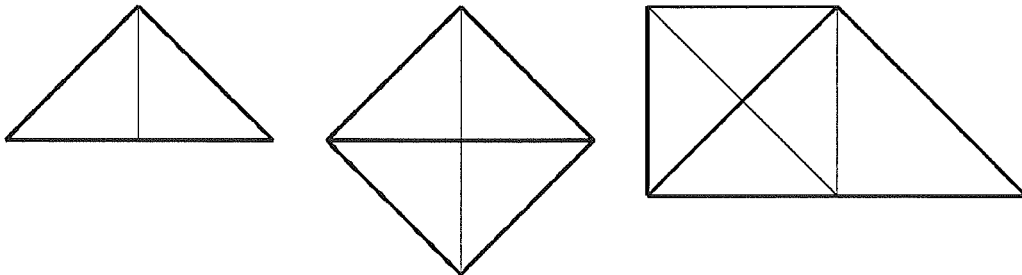


Figure 4: Examples of split types. Single Split (left) Diamond Split (center) Split Recursion (right)

All of the above split rules, if applied, prevent the formation of T-vertices. Since there are no T-vertices to begin with, this ensures that there are no T-vertices in the triangle mesh. This fulfils the first condition from Section 5.

5.2 A Parallel Algorithm for Adaptive Binary Triangle Tree Construction

Constructing a binary triangle tree is a difficult task to parallelize, as it is difficult to break it into a set of independent tasks. This is because any split may force a neighboring triangle to

split which makes triangles dependant on their neighbors. Therefore, they cannot be treated as individual and independent objects.

Simply splitting triangles in parallel without regard for mutual exclusion will lead to a race condition. To avoid multiple threads modifying the same triangle simultaneously, the obvious approach would be to have a mutex for each triangle. An implementation of this approach using OpenMP under C++ using the Intel C++ Compiler showed that the parallel overhead involved with creating, acquiring, and releasing the locks exceeded the runtime of the algorithm with parallelism disabled.

A more efficient approach is to partition the parameter space into sections, construct an independent binary triangle tree for each of these sections and, then, merge the sections together. This approach is the basis of the algorithm presented below.

The algorithm is broken into three distinct phases. The last two phases are run in parallel, the first one not. Synchronization, namely a barrier, is required between the phases to ensure correctness.

The phases are as follows:

1. The first phase is the initialization phase. This involves the partitioning of the parameter space and needs to run serially.

To ensure that the end result of the algorithm is the same as that of the serial algorithm, it is important that the partitioning itself forms binary triangle trees. For ease of implementation, the partitioning used in this report also ensures that no T-vertices are present at this stage.

Load balancing is an issue that needs consideration. Since the sections will be distributed amongst the threads, if some threads are allocated more work than others, idle waiting may become a performance issue.

It is important to ensure that the sections are of a fairly even size with respect to the amount of processing that needs to be done. It is possible that bad partitioning may place almost all the work in very few sections and leave many sections with very little to do.

While determining the exact amount of work that a section requires is often difficult to calculate, the error metric g (see Section 5.1) provides an approximation.

In this report, for the purposes of partitioning, a minimum number of sections is set and the serial binary-triangle-tree-building algorithm from Section 5.1 is run, until at

least that many triangles are in the mesh (or all triangles satisfy the accuracy test, in which case the algorithm finishes early). There is one difference, namely that the algorithm is altered so that it splits less accurate triangles first; this can be easily achieved using a priority queue which uses g as a comparison function.

This approach prevents triangles with a low level of inaccuracy from being further partitioned and ensures triangles with a high level of inaccuracy are partitioned first. The triangles left when the algorithm finishes become the sections to be processed by the later stages.

To further ensure even load balancing, a large number smaller sections should be used. Smaller sections should take less time to process than larger ones and hence the amount of possible idle waiting time is decreased as tasks become smaller. However, if there are too many sections, then the time taken for partitioning increases; this will decrease the proportion of work that can be done in parallel. Also, too many sections will lead to an increase in the number mutexes needed for synchronization. This means that a balance must be found that gives the optimal value for the minimum number of sections passed to the partitioning algorithm.

2. In the second phase the sections are individually processed. The binary-triangle-tree-building algorithm is simply run once on each section. This can be done in parallel and the sections can be distributed amongst the threads in a thread pool like manner.
3. The third phase involves joining the sections together to form a single coherent binary triangle tree and triangle mesh. Since the sections are in two binary triangle trees and each section forms a binary triangle tree, it is easy to create two binary triangle trees for the whole mesh, simply replacing each section in the 'section trees' with its binary triangle tree.

Joining the sections into one large triangle mesh will create a triangle mesh which fulfils the second and third criteria mentioned in Section 5. However, such a triangle mesh might have T-vertices where the sections join up.

The third stage of the parallel algorithm is primarily concerned with removing T-vertices from the borders of sections. Each border can be considered as a separate task, therefore, the borders can be distributed amongst the threads for processing. However, when processing a border, a thread needs exclusive access to both sections sharing that border which can be achieved through mutexes.

Joining borders can be done recursively. The triangle sides along a border may be split, if one side of the border is split and the other is not, then a split needs to be done to ensure that they match up and that there are no T-vertices. Where the border is split,

the triangles can be joined recursively along the two halves of the border. This forms the basis of the joining algorithm shown in Section 8.1 (in the appendix).

Special care must be taken when joining two triangles along one border affects triangles along another border. This may happen when splitting a triangle results in a triangle that is on another border being split also. Splitting a triangle may cause another split along a border that needs to be joined, in this situation, the border that has been modified needs to be re-joined.

Once all the borders have been joined (and possibly re-joined), the third phase and the algorithm are complete. A pair of binary triangle trees equivalent to those produced by the serial algorithm will have been produced.

This algorithm has been implemented and its performance is discussed in Section 5.2.1.

5.2.1 Performance of the Parallel Algorithm

The above algorithm was implemented using C++ and OpenMP, compiled using the Intel C++ Compiler and run on an eight Core Intel Quad-Core Xeon E5335 (2GHz) computer. The following table shows the results.

The algorithm was run three times with a tolerance of 3.5×10^{-9} and five times for the other tolerance levels. The minimum and maximum observed times (in seconds) are shown. The minimum time is used in the speedup calculation. The number of triangles in the mesh generated at each tolerance level is also shown. It was ensured that sufficient physical memory was available at all times during the execution of the algorithm. The C `time.h` library's `clock()` function was used for timing, as such, start and finish times were recorded to the nearest millisecond. Simple mathematical functions were used for the surface and the error metric.

	Tolerance	#Triangles		Tolerance	#Triangles	
	1×10^{-6}	213616		1×10^{-7}	2155744	
	Min/s	Max/s	Speedup	Min/s	Max/s	Speedup
Serial	0.124	0.140	1	1.218	1.484	1
1 Thread	0.124	0.140	1.000	1.218	1.234	1.000
2 Threads	0.093	0.109	1.333	0.765	0.828	1.592
3 Threads	0.062	0.094	2.000	0.609	0.656	2.000
4 Threads	0.062	0.078	2.000	0.499	0.531	2.441
5 Threads	0.062	0.078	2.000	0.499	0.515	2.441
6 Threads	0.047	0.078	2.639	0.515	0.546	2.365
7 Threads	0.046	0.078	2.696	0.515	0.516	2.365
8 Threads	0.047	0.124	2.638	0.531	0.546	2.294

	Tolerance	#Triangles		Tolerance	#Triangles	
	1×10^{-8}	21650816		3.5×10^{-9}	61796376	
	Min/s	Max/s	Speedup	Min/s	Max/s	Speedup
Serial	12.047	12.062	1	35.171	35.547	1
1 Thread	12.312	12.327	0.978	35.390	35.484	0.994
2 Threads	7.812	8.249	1.542	23.437	23.530	1.501
3 Threads	6.328	6.468	1.904	18.421	18.671	1.909
4 Threads	5.141	5.469	2.343	15.140	15.827	2.323
5 Threads	5.124	5.203	2.351	14.562	14.891	2.415
6 Threads	5.281	5.375	2.281	15.312	15.593	2.297
7 Threads	5.234	5.281	2.302	14.922	15.046	2.357
8 Threads	5.453	5.546	2.209	15.593	15.796	2.256

The results show that the parallel algorithm does offer an improvement over the serial algorithm as far as performance is concerned. It is clear that three threads are sufficient to nearly double the speed of the algorithm and that having more than about four threads offers little or no increase in performance.

The speedup shown by the parallel binary-triangle-tree-building algorithm is not quite as impressive as that shown by the parallel Cholesky decomposition algorithm from Section 4.1. This shows that, even though the binary-triangle-tree-building algorithm is not naturally suited to parallelism and requires significant alteration to parallelize, the algorithm can benefit from parallelism.

6 Conclusion

This report has, among other things, demonstrated three key points important in parallel computing.

- There is a wide range of difficulty in parallelizing algorithms, difficulty ranges from the very easy like the Cholesky decomposition algorithm and the forwards and backwards substitution algorithms to the complex like the binary-triangle-tree-building algorithm. The key to parallelizability is to divide the overall task into a set of independent tasks that can be executed in any order, the difficulty in parallelization arises from finding these tasks.

As was seen in the cases of the Cholesky decomposition algorithm and the forwards and backwards substitution algorithms, these independent tasks may already exist in the serial algorithm. In cases like these, parallelization is merely a matter of identifying the

independent tasks and then ensuring that they are run in parallel. When using a compiler directive extension like OpenMP, this can be achieved in only a few lines of code.

On the other end of the spectrum, the serial binary-triangle-tree-building algorithm lacks independent sub-tasks. Here the independent sub-tasks need to be created by modifying the algorithm itself. The algorithm that was presented in Section 5.2 showed how complex parallelization can become. In designing the algorithm, all the issues raised by mutexes, barriers, race conditions, load balancing and whether or not the parallel algorithm produces correct output had to be considered.

- Significant performance improvements can be achieved through parallelism. Additionally, it has been shown that, even if parallelization is difficult, as was the case with the binary-triangle-tree-building algorithm, the benefits may still be worthwhile.
- The simple and obvious approach is not always the best. In the binary-triangle-tree-building algorithm, simply using a mutex for every triangle proved to be much less efficient than a more complex partitioning approach. This demonstrated the well known effects of granularity¹².

7 Bibliography

References

- [Chandra et al., 2001] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- [Duchaineau et al., 1997] Duchaineau, M., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., and Mineev-Weinstein, M. B. (1997). Roaming terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 81–88, Los Alamitos, CA, USA. IEEE Computer Society Press.
- [Evans et al., 1996] Evans, F., Skiena, S., and Varshney, A. (1996). Optimizing triangle strips for fast rendering. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 319–326, Los Alamitos, CA, USA. IEEE Computer Society Press.
- [Golub and van Loan, 1996] Golub, G. H. and van Loan, C. F. (1996). *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA.
- [Intel, 2007] Intel (2007). Intel processor pricing. http://www.intel.com/intel/finance/pricelist/processor_price_list.pdf?iid=search Accessed January 2008.
- [Pressel et al., 1999] Pressel, D. M., Behr, M., and Thompson, S. (1999). The true limitations of shared memory programming. In Arabnia, H. R., editor, *PDPTA*, pages 2048–2054. CSREA Press.

¹²Granularity was briefly discussed in Section 3.1.2, more detail can be found in [Chandra et al., 2001] pages 172 to 175.

[Shonkwiler and Lefton, 2006] Shonkwiler, R. W. and Lefton, L. (2006). *An Introduction to Parallel and Vector Scientific Computation (Cambridge Texts in Applied Mathematics)*. Cambridge University Press, New York, NY, USA.

[Tanenbaum and Goodman, 1998] Tanenbaum, A. S. and Goodman, J. R. (1998). *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

8 Appendix

8.1 Join Function

The following is pseudo-code for a function that can be used in phase three of the algorithm described in Section 5.2. Its purpose is to assist in removing T-vertices from a triangle mesh constructed using binary triangle trees. It is included as it was reasonably difficult to design correctly.

The algorithm described below joins two triangles which share a specified side. The side, vertex and child numbering is consistent with that in the reference diagram in Figure 1. (Note that the pseudo-code assumes triangles are passed by reference in a manner similar to Java.)

```

1      function join(Triangle tri1, int side1, Triangle tri2, int side2)
2          if side1 > side2 then
3              swap(tri1, tri2)
4              swap(side1, side2)
5          end
6          (Now we can assume side1 <= side2 without loss of generality)
7          if side1 = 0 & side2 = 0 then
8              if tri1.isChildLess & tri2.isNotChildLess then tri1.split()
9              if tri1.isNotChildLess & tri2.isChildLess then tri2.split()
10             if tri1.isNotChildLess then
11                 join(tri1.child1, 1, tri2.child2, 2)
12                 join(tri1.child2, 2, tri2.child1, 1)
13             end
14         end
15         if side1 = 0 & side2 = 1 then
16             if tri1.isNotChildLess & tri2.isChildLess then tri2.split()
17             if tri2.isNotChildLess then
18                 join(tri1, 0, tri2.child2, 0)
19             end
20         end
21         if side1 = 0 & side2 = 2 then
22             if tri1.isNotChildLess & tri2.isChildLess then tri2.split()
23             if tri2.isNotChildLess then
24                 join(tri1, 0, tri2.child1, 0)
25             end

```

```

26         end
27         if side1 = 1 & side2 = 2 then
28             if tri1.isNotChildLess then
29                 if tri2.isNotChildLess then
30                     join(tri1.child2, 0, tri2.child1, 0)
31                 else
32                     if tri1.child2.isNotChildLess then
33                         tri2.split()
34                         join(tri1.child2, 0, tri2.child1, 0)
35                     end
36                 end
37             else
38                 if tri2.isNotChildLess & tri2.child1.isNotChildLess then
39                     tri1.split()
40                     join(tri1.child2, 0, tri2.child1, 0)
41                 end
42             end
43         end
44         (Other cases should not occur in this algorithm)
45     end

```

On completion of this function there will be no T-vertices on the border between `tri1` and `tri2`.

Note that the `sideX` parameters are meant to identify how the two triangle parameters relate, `tri2` is on side `side1` of `tri1` and `tri1` is on side `side2` of `tri2`.

8.2 Triangle Strips and Fans

This section is an extension of the section on binary triangle trees. It studies an important task related to triangle meshes and, therefore, binary triangle trees. It is included as an aside.

When a polygonal mesh, created by an algorithm such as the one presented in Section 5, is displayed on a computer, optimization relating to the display process itself becomes important. Polygons are passed to the computer graphics hardware as a sequence of vertices that define the relevant polygons. Each vertex that is passed to the graphics hardware requires processing to determine its color and position on the display.

A triangle has three vertices; the simplest approach is to pass three vertices per triangle to the graphics hardware. However, in a triangle mesh most vertices are shared by multiple triangles. As such, it is inefficient to process the same vertex once for every triangle that shares it.

Triangle strips and triangle fans are a means of reusing already processed vertices in further triangle definitions. While there are other means of achieving this these are two of the most

common. In a triangle strip or fan the first triangle is defined, as one would expect, with its three vertices. However, all the following triangles are defined with only one additional vertex; this is made possible by ‘reusing’ two vertices used in the definition of previous triangles. The difference between a triangle strip and a fan is which vertices are reused; a triangle strip reuses the two most recent vertices to be passed and a triangle fan reuses the most recent vertex and the first vertex.

Figure 5 shows an example of a triangle strip and a triangle fan.

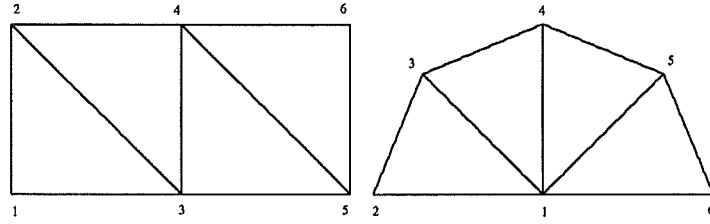


Figure 5: An example of a triangle strip. (left) An example of a triangle fan. (right) The vertices are numbered in the order they are passed to the graphics hardware.

If there are n triangles in a given triangle mesh, then $3n$ vertices need to be processed if each triangle is defined using three vertices. However, if there are k strips or fans¹³, then the number of vertices needing to be processed¹⁴ is $2k + n$. Clearly:

$$2k + n \leq 3n, \quad \forall n \geq k \geq 0;$$

also, the average length¹⁵, l , of a triangle strip or fan is:

$$l = \frac{n}{k}.$$

This means that the number of vertices needing to be processed for the triangle mesh is:

$$2\frac{n}{l} + n$$

So, using triangle strips or fans is at least as efficient as using three vertices per triangle (in terms of the number of vertices needing to be processed). Also, the longer the triangle strips or fans are on average, the fewer vertices need to be processed. This leads to the conclusion that representing a triangle mesh as a set of triangle strips or fans is better than representing it as a set of individual triangles.

¹³Note that individual triangles are considered as a triangle strip or fan containing one triangle.

¹⁴Note that the number of vertices needing processing for a given number triangles is the same for both triangle strips and fans. Both are equally efficient in this respect.

¹⁵The length is defined as the number of triangles represented by the triangle strip or fan.

Taking a triangle mesh represented as individual triangles and joining them into strips or fans of maximal average length is a difficult problem. Normally a heuristic of some form is used to create triangle strips or fans. (See [Evans et al., 1996] for further examination of triangle strip generation.)

In [Duchaineau et al., 1997] a heuristic was proposed for creating long triangle strips as part of the process of constructing the binary triangle tree. It involved incrementally adding and removing triangles into and from strips as they were being split. According the paper this approach “yields average strip lengths of around four to five triangles” (section 7.2, page 7 of [Duchaineau et al., 1997]).

This section investigates the performance of an alternative triangle fan based heuristic.

There are some important points to note about triangle fans in meshes constructed using the binary triangle tree algorithm. They are:

1. Every triangle fan must contain an integral number of triangles (as non-integral numbers of triangles and vertices are absurd).
2. Every triangle fan must contain at least one triangle (otherwise there would be no need to represent it at all).
3. Since each triangle in the mesh is a right angled isosceles triangle, every corner of a triangle has an angle of at least $\frac{\pi}{4}$ (in parameter space).
4. Since there is a maximum available angle of 2π around any point in the parameter space plane, due to point 3 there can be no more than eight triangles surrounding any vertex.
5. Since each triangle in a triangle fan shares a common vertex, the triangles must surround that vertex.
6. Due to points 4 and 5 no triangle strip can contain more than eight triangles.
7. Due to points 1, 2 and 6 every triangle fan must be of length one, two, three, four, five, six, seven or eight.

Since longer triangle fans are desired, the heuristic begins by searching for fans of the maximum possible length, namely eight; all of these fans are added to the set of fans to be displayed and the triangles in it flagged so that they are not included in further fans. The heuristic then tries searching for fans of length seven, then six and so on until it reaches one, at which point all triangles will be included in a fan.

The basic pseudo-code for this is:

```

1    $S \leftarrow \{\}$ 
2   for i = 8 to 1
3       for all x in {unmarked triangles}
4           if x is part of a fan of unmarked triangles of length i then
5               mark x
6               add the triangle fan to S
7           end
8       end
9   end
10  display S

```

Clearly this heuristic is easy to parallelize alongside the parallel binary-triangle-tree-building algorithm from Section 5.2. The sections can be independently processed by partitioning the set of unmarked triangles using the sections and then once again distributing the sections amongst threads.

The algorithm was implemented and tested on some binary triangle tree meshes, the average length of fans produced varied from 4 to 5.7. This shows that a triangle fan approach yields results similar to a triangle strip approach.